A TRANSFORMATIONAL APPROACH TO NEGATION IN LOGIC PROGRAMMING*  

ROBERTO BARBUTI, PAOLO MANCARELLA, DINO PEDRESCHI, AND FRANCO TURINI

A transformation technique is introduced which, given the Horn-clause definitions of a set of predicates \( p_i \), synthesizes the definitions of new predicates \( \bar{p}_i \) which can be used, under a suitable refutation procedure, to compute the finite failure set of \( p_i \). This technique exhibits some computational advantages, such as the possibility of computing nonground negative goals still preserving the capability of producing answers. The refutation procedure, named SLDN refutation, is proved sound and complete with respect to the completed program.

1. INTRODUCTION

Negation as failure has been thoroughly studied as a means to deal with negative information in logic programming [8, 25–27]. It is a meta-inference-rule allowing one to prove the negation of a ground goal, when the proof of the corresponding positive goal finitely fails. Starting from Clark’s paper [8], a lot of effort has been devoted to establishing results about soundness and completeness of this rule with respect to completed logic programs [8, 9, 28], and to extending these results to subclasses of general logic programs, i.e. programs containing negative literals in clause bodies [3, 5, 7, 16, 17, 26].

The major drawback of a logic-programming system embodying negation as failure is that it does not allow a symmetric treatment of positive and negative knowledge. In fact, negation as failure can be used only to check universally quantified negative formulae, and by no means can it be used to compute solutions for existentially quantified negative ones. Indeed, in many applications, such as

*Work partially supported by M.P.I., 40%.

Address correspondence to Paolo Mancarella, Dipartimento di Informatica, Università di Pisa, Corso Italia, 40, 56100 Pisa, Italy.

Received December 1987; accepted June 1988.
deductive databases and expert systems, it would be useful to compute answers both for positive and negative queries in a homogeneous way.

This paper presents a transformational approach to the problem of negation, aimed at meeting the above requirement by exploiting and formalizing the ideas sketched in [6], which in turn can be regarded as an extension of the techniques of [23] to the full class of logic programs.

Given a logic program $P$, our transformation technique is able to synthesize the definition of a new predicate $\tilde{p}$ for each predicate $p$ defined in $P$, the intended meaning of $\tilde{p}$ being, roughly speaking, the negation of $p$. Unfortunately, the definition of the negated predicates is not, in general, a set of Horn clauses, and hence a suitable refutation procedure, called SLDN refutation hereafter, has to be devised to cope with them. It will be shown that the (SLDN) success set of the negated program coincides with the (SLD) finite failure set of the original one. More importantly, SLDN resolution is actually able to compute answer substitutions for nonground goals involving the negated predicates, thus providing solutions for (existentially quantified) negative queries with respect to the original program. As a consequence, the approach is semantically equivalent to negation as failure, although strictly more expressive from a computational viewpoint.

The paper is organized as follows: Section 2 summarizes some basic results concerning negation as failure and sets up some preliminary definitions and notation; Sections 3 and 4 deal with the formal definition of the transformation. Section 5 introduces SLDN resolution, and Section 6 states the results about its soundness and completeness with respect to negation as failure. Finally, Section 7 gives an alternative definition of the transformation, yielding more concise negated programs, which is harder to tackle formally but easier to implement.

2. PRELIMINARIES

Let us first summarize some known results about negation in logic programming. A thorough presentation of this subject can be found in [15,26]. As pointed out in the introduction, most logic-programming systems adopt the negation-as-failure rule. Negation as failure can be viewed as an effective approximation of the closed-world assumption (CWA) [22], which states that if a ground atom $A$ is not a logical consequence of a program $P$, then $\neg A$ is inferred.

Under negation as failure, it is possible to infer $\neg A$ if and only if, for any fixed fair computation rule, $A$ has only a finite number of finite SLD derivations, which all fail. Since the finite failure of an atom $A$ is a way to state in a finite amount of time that $A$ is not a logical consequence of $P$, then negation as failure is actually a computable approximation of the CWA (recall that SLD resolution is semidecidable).

Negation as failure has been proved correct and complete with respect to the completed program $(P^*, U^*)$ introduced in [8] (the notation $(P^*, U^*)$ is borrowed from [10]). Informally, $P^*$ is a transformation of the original program $P$ which introduces also the “only if” part of the predicate definitions. On the other hand, $U^*$ is the equality theory underlying unification, which gives a precise meaning to the equality predicate used in $P^*$. According to [2,10], let $\text{FF}(P)$ be the set of all $A \in B(P)$ (the Herbrand base of $P$) such that a finitely failed SLD tree exists for
The soundness result states that if $A \in \text{FF}(P)$, then $\neg A$ is a logical consequence of $(P^*, U^*)$. On the other hand, the completeness result states that if $\neg A$ is a logical consequence of $(P^*, U^*)$, then $A$ is in $\text{FF}(P)$. Moreover, using a fair SLD resolution [13], a sound and complete implementation of $\text{FF}(P)$ is obtained.

In [18] a stronger completeness result for negation as failure is stated, referring to completed programs $(P^*, U^*_{\text{DCA}})$, where $U^*_{\text{DCA}}$ stands for the equality theory $U^*$ augmented with the domain closure axiom (DCA: the terminology is borrowed from [16]). Roughly speaking, the effect of DCA is to constrain the models of $(P^*, U^*_{\text{DCA}})$ to be closer (although not isomorphic) to Herbrand models. In the sequel it will be clear that DCA plays a central role in the proposed transformation technique.

Let us now introduce some notation. Throughout the paper we will refer to an underlying first-order language $L$ which provides a set of constructor symbols $\{c_i\}_{i=1,\ldots,n}$, each of arity $k_i \geq 0$. When $k_i = 0$, $c_i$ is a constant symbol; otherwise it is a function symbol. $L$ is supposed to provide at least one constant. The Herbrand universe associated to $L$ is denoted by $H_L$. We will use $t, u, s, \ldots$ (possibly indexed) to range over terms of $L$, and $x, v, w, y, z, \ldots$ to denote variables; when needed, a tuple of terms will be denoted by $\bar{t}, \bar{u}, \ldots$, and similarly for tuples of variables; the notation $\bar{t}^{(\bar{x})}$ ($\bar{t}^{(\bar{z})}$) is used to indicate that the tuple $\bar{t}$ contains only variables in $\bar{x}$ (in $\bar{z}$ and $\bar{y}$). Many examples will refer to the language Nat with constant 0 and function $s$, of arity one.

As in [14], the following terminology will be adopted. A term $t$ is called restricted if at least one variable occurs in it more than once, unrestricted otherwise. For example $c_1(x, x)$ is restricted, whereas $c_1(x, y)$ is unrestricted. This notion trivially generalizes to tuples of terms. The notation $\bar{s} \leq \bar{t}$ will be used to indicate that $\bar{s}$ is an instance of $\bar{t}$, i.e., there exists a substitution $\gamma$ such that $\bar{s} = \bar{t} \gamma$. In a logic program $P$ over $L$, the letters $p, q, r, \ldots$ will be used for predicate symbols. A logic program $P$ is left-linear if each clause head contains an unrestricted tuple of terms. Given a formula $A$ with free variables $\bar{x}$, the shorthand $\exists \bar{x}. A$ ($\forall \bar{x}. A$) will often be used instead of $\exists \bar{x}. A$ ($\forall \bar{x}. A$).

### 3. Computing the Complement of Unrestricted Terms

A key point in the transformation technique defined in the next section is the ability to compute the complement of a given term $t$. The idea is that a term $t$ can be viewed as an intensional representation of the set of its ground instances with respect to some interpretation domain. Hence, its complement should represent the set-theoretic complement of such a set. In order to do this, we first define formally an equality theory which precisely characterizes what the interpretation domains should look like, and secondly give an algorithm which computes the complement of a term $t$.

The equality theory we will refer to throughout the paper is actually an extension of Clark's equality theory, similar to the one defined in [16, 17]. Referring to the underlying language, $L$, $U^*_{\text{DCA}}$ stands for such a theory, which contains the following axioms:

1. $\forall x(x = x)$.
2. $\forall yzw((x = y \land z = w) \rightarrow (x = z \rightarrow y = w))$. 


(3) $\forall \bar{x} \bar{z} (\bar{x} = \bar{z} \Leftrightarrow c_i(\bar{x}) = c_i(\bar{z}))$, $c_i$ a function symbol and $\bar{x}, \bar{z}$ tuples of length $k_i$.

(4) $\forall \bar{x} \bar{z} (c_j(\bar{x}) \neq c_i(\bar{z}))$, $i, j = 1, \ldots, k$ and $i \neq j$.

(5) $\forall x (t_{[\bar{z}]} \neq x)$, $t_{[\bar{z}]}$ a nonvariable term, $x$ in $\bar{z}$.

(6) $\forall x (\forall i = 1, \ldots, k \exists \bar{z} (x = c_i(\bar{z})))$.

(1) is the axiom for symmetry of $=$, while (2) is the axiom for both reflexivity and transitivity. (3) ensures that each function symbol must be interpreted as a function ("if" part) and also that this function must be injective ("only if" part). (4) ensures that objects constructed using different leftmost constructors are different, that is, each function (each constant) symbol has to be interpreted as a distinct function (object). (5) states that if $t$ is a subterm of $t'$, then $t$ and $t'$ must be mapped into different objects: This corresponds to the so-called occur check in the unification algorithm. Finally, (6) is called in [16] the domain closure axiom (DCA). It forces the interpretation domain of a model to contain only objects which can be constructed (even if not effectively) using the (interpretation of) constant and function symbols. In the sequel, the superscript $L$ in $U^*_L$ will be omitted when no confusion can arise.

In [18] a result characterizing the models of $U^*_D$ is given, which can be summarized as follows.

Proposition 3.1. Let $M_D$ be a model of $U^*_D$ over the domain $D$. Then

(a) $D$ contains an isomorphic copy $H'$ of the Herbrand universe $H_L$, and $=$ restricted to $H'$ is the identity relation;

(b) $d \in D \setminus H'$ iff $d = c_i(d_1, \ldots, d_m)$ for some $c_i$ and at least one $d_j \in D \setminus H'$.

Proposition 3.1 precisely characterizes the fact that DCA is actually an approximation of the domain closure assumption [24]. In fact the latter would force interpretations to be Herbrand interpretations only, but it is not first-order definable, whereas the former forces interpretation domains to contain only well-formed objects. The following example is an instantiation of $U^*_D$ when $L$ is the language Nat, yielding the $U^*_D$ theory.

Example 3.1

(1) $\forall x (x = x)$.

(2) $\forall xyzw ((x = y \land z = w) \rightarrow (x = z \rightarrow y = w))$.

(3) $\forall xz (x = z \leftrightarrow s(x) = s(z))$.

(4) $\forall x (s(x) \neq 0)$.

(5) $\forall x (s^n(x) \neq x)$, $n \geq 1$.

(6) $\forall x (x = 0 \lor \exists z. x = s(z))$.

The set $N$ of natural numbers, equipped with the obvious mapping for $0$ and $s$ and with the interpretation of $=$ as the identity relation, is of course a model of $U^*_D$. On the other hand, $N \cup \{ \cdot \}$, with $\cdot$ distinct from any natural number, is by no means a model of $U^*_D$, if $\cdot$ is not constructed via $s$ from some other object.
Take instead the disjoint union $N + Z$, where $Z$ is the set of (positive and negative) integers (disjoint union avoids confusion between naturals and nonnegative integers). Moreover, define an appropriate successor function $\text{Succ}$ on $N + Z$, which yields either the natural or integer successor depending on the argument. Mapping 0 into 0 $\in N$, $s$ into the function $\text{Succ}$, and finally $=\in N \times N \cup Z \times Z$ yields a model of $U_{DCA}^{*\text{Nat}}$. Notice that there is no way to effectively denote an object in $Z$ by means of a ground term of the language, even if an open term like $s(x)$ has objects in $Z$ as instances. Finally, notice that this model satisfies DCA, since each $m \in Z$ is well formed, namely, there exists an object $z \in Z$ such that $m = s(z)$.

The next step is to compute the complement of a term $t$ within the $U_{DCA}^{*}$ theory. As mentioned before, the basic idea is that a term $t$ can be viewed as an intensional representation of a set with respect to an interpretation domain $D$ of a model of $U_{DCA}^{*}$. For instance, referring again to the language $\text{Nat}$ and to the interpretation domain $N + Z$ as in Example 3.1, the term $s(x)$ represents the set $\{n \in N \mid n \geq 1\} \cup Z$ (throughout this section we will improperly denote the set represented by a term $t$ by the term $t$ itself, since the context will remove this ambiguity).

It can be shown [1] that the set-theoretic complement of an unrestricted term $t$ can be intensionally represented as well, by means of a finite set of unrestricted terms. For example, the set-theoretic complement of $s(x)$ is the set $\{0\}$, and the complement of $s(0)$ is $\{0, s(s(x))\}$. Moreover, such a set can be effectively computed by an algorithm, which is introduced as a recursive function $\text{Not}_L$.

**Definition 3.1.** The function $\text{Not}_L$ which, given an unrestricted term $t$, yields the set $\{t_1, \ldots, t_k\}$ corresponding to its set-theoretic complement is defined by the following three rules:

1. $(\text{Not}_1)\, \text{Not}_L(x) = \{\}$.
2. $(\text{Not}_2)\, \text{Not}_L(c_j) = \{c_j \mid c_j \text{ a constant symbol, } c_j \neq c_i\} \cup \{c_j(\bar{x}) \mid c_j \text{ function symbol}\}$ if $c_i$ is a constant.
3. $(\text{Not}_3)\, \text{Not}_L(c_i(t_1, \ldots, t_m)) = \{c_j \mid c_j \text{ a constant symbol}\} \cup \{c_j(\bar{x}) \mid c_j \text{ a function symbol, } c_j \neq c_i\} \cup \{c_i(x_1, \ldots, x_{k-1}, t', x_{k+1}, \ldots, x_m) \mid k = 1, \ldots, m, t' \in \text{Not}_L(t_k)\}$.

It is worth noting that the variables introduced in the resulting terms are fresh ones and are distinct each other. Moreover, $\text{Not}_L$ trivially extends to operate on unrestricted tuples of terms (actually this is already embedded in the rule $\text{Not}_3$).

**Example 3.2.**

$$\begin{align*}
\text{Not}_{\text{Nat}}(x) &= \{\}, \\
\text{Not}_{\text{Nat}}(0) &= \{s(y)\}, \\
\text{Not}_{\text{Nat}}(s(t)) &= \{0\} \cup \{s(t') \mid t' \in \text{Not}_{\text{Nat}}(t)\}.
\end{align*}$$

Hence, the complement of $t = s(0)$ is computed as

$$\text{Not}_{\text{Nat}}(s(0)) = \{0\} \cup \{s(t') \mid t' \in \text{Not}_{\text{Nat}}(0)\} \quad \text{(by Not}_3)$$

$$= \{0, s(s(y))\} \quad \text{(by Not}_2).$$
Example 3.3. Let $L$ be a language introducing constants $a, b$ and the binary function $f$. Then

\[
\begin{align*}
\text{Not}_L(x) &= \{ \} , \\
\text{Not}_L(a) &= \{ b, f(x, y) \} , \\
\text{Not}_L(b) &= \{ a, f(x, y) \} , \\
\text{Not}_L(f(t, s)) &= \{ a, b \} \cup \{ f(t', x) | t' \in \text{Not}_L(t) \} \cup \{ f(x, s') | s' \in \text{Not}_L(s) \} .
\end{align*}
\]

Hence

\[
\text{Not}_L(f(f(a, x), b)) = \{ a, b, f(a, x), f(b, x), f(f(b, x), y), f(f(f(x, y), z), w), f(x, a), f(x, f(y, z)) \} .
\]

It should be stressed that the Not algorithm does not work properly when applied to restricted terms, as pointed out in the following

Example 3.4. With respect to the language $L$ of Example 3.3, $\text{Not}_L(f(x, x))$ yields $\{ a, b \}$, whereas the actual complement of $f(x, x)$ is $\{ a, b \} \cup \{ f(x, y) | x \neq y \}$. As remarked in [14], the complement of a restricted term cannot be represented by a finite disjunction of terms.

In the sequel we will refer to $\text{Not}(\bar{t})$ as the complement of a tuple $\bar{t}$. The following theorem points out the basic property of the Not algorithm, that is, its correctness with respect to the theory $U_{DCA}^*$. Theorem 3.1 is closely related to the algorithm uncover and its correctness proof in [14].

Theorem 3.1. Let $\bar{t}$ be an unrestricted tuple of terms, $\{ \bar{t}_1, \ldots, \bar{t}_m \} = \text{Not}_L(\bar{t})$, and $\bar{x}$ a tuple of fresh variables. Then

\[
U_{DCA}^* \models \forall \bar{x} \left( \forall \bar{x} \neq \bar{t} \iff \bigvee_{i=1, \ldots, m} \exists \bar{x} = \bar{t}_i \right).
\]

Proof. For the sake of simplicity we prove the result when $\bar{t}$ is a tuple of one element $t$. The proof itself trivially generalizes to $n$-tuples of terms. Let $M_D$ be an arbitrary model of $U_{DCA}^*$ over a domain $D$. Actually, we prove, by structural induction on $t$, that for each $d \in D$, $d$ is an instance of $t$ if and only if it is not an instance of any $t_i, i \in [1, m]$. Let $d = [t]$ stand for "$d$ is an instance of $t$ (over a given domain $D$)".

Base step: If $t$ is a variable then any $d \in D$ is an instance of $t$ and $\text{Not}_L(t)$ is empty. Otherwise, if $t$ is the constant $c_i$, then

\[
\text{Not}_L(c_i) = \{ c_j | c_j \text{ a constant symbol, } c_i \neq c_j \} \cup \{ c_j(\bar{x}) | c_j \text{ a function symbol} \} .
\]

Take an arbitrary $d \in D$. If $d = [c_j]$, $c_j$ a constant symbol, then the proposition holds trivially. If $d = [c_j(\bar{a})]$, then $d$ is an instance of $c_j(\bar{x})$ which belongs to $\text{Not}_L(c_j)$ and again the proposition holds. No other cases on $d$ must be taken into consideration as base step, because of Proposition 3.1.

Inductive step: Let $t = c_j(t_1, \ldots, t_k)$. Take an arbitrary $d \in D$. If $d = [c_j]$ or $d = [c_j(\bar{a})]$, with $j \neq i$, then, as in the base step, $d$ is not an instance of $t$ and it is an instance of a term in $\text{Not}_L(t)$, either the term $c_j$ or the term $c_j(\bar{x})$. Suppose
By the induction hypothesis, \( \alpha_h \) is an instance of \( t_h \) iff \( \alpha_h \) is not an instance of any \( u \in \text{Not}_L(t_h) \), for each \( h \in [1, k] \). Hence, if for each \( h \in [1, k] \) \( \alpha_h \) is an instance of \( t_h \), then \( d \) is an instance of \( t \) and by no means can it be an instance of a term in \( \text{Not}_I(t) \), since each term in \( \text{Not}_I(t) \) with leftmost constructor \( c_i \) contains as a subterm a term \( u \in \text{Not}_I(t_h) \), for each \( h \in [1, k] \). Conversely, if for some \( h \in [1, k] \) \( \alpha_h \) is an instance of some \( u \in \text{Not}_I(t_h) \), then \( d \) is an instance of the term \( c_i(x_1, \ldots, x_{k-1}, u, x_{k+1}, \ldots, x_k) \in \text{Not}_I(t) \), and it is not an instance of \( t \), since by the induction hypothesis \( d_h \) is not an instance of \( t \). Since \( t \) is unrestricted and, again, Proposition 3.1 holds, no other cases of \( d \) need be taken into consideration in the inductive step. □

Notice that, on dropping DCA from the equality theory, the above result does not hold any more. As an example, referring to the language Nat, consider the term 0 and its complement \( s(y) \). Then \( U_{DCA}^{\text{Nat}} \neq \forall x (x \neq 0 \leftrightarrow \exists y. x = s(y)) \), but this is definitely false on getting rid of DCA, since then the fact that an object is not 0 does not imply that this object is constructed by means of the constructors.

**4. NEGATING LOGIC PROGRAMS**

In this section the transformation technique for negating logic programs is introduced. In order to simplify the transformation itself, it will be defined for a special class of logic programs, named *factorized logic programs* (from now on FLPs).

**Definition 4.1.** An FLP is a logic program where each predicate \( p \) either

- is the predicate eq, which is defined by the only clause eq(\( x, x \)) :- , or
- is defined according to one of the following cases:
  
  (i) \( p(\overline{x}) :- q(\overline{x}), r(\overline{x}) \),
  
  (ii) \( p(\overline{x}) :- q(\overline{x}); r(\overline{x}) \),
  
  (iii) \( p(\overline{x}) :- q(\overline{x}, \overline{y}) \),
  
  (iv) \( p(i) :- q(i') \),

  where in cases (i), (ii), (iii) \( \overline{x} \) and \( \overline{y} \) stand for tuples of distinct variables. In case (iv) \( i \) stands for an unrestricted tuple of terms such that \( \text{vars}(i) \supseteq \text{vars}(i') \), and the literal \( q(i') \) may be omitted (in this case the clause is a fact rule).

Notice that FLPs, apart from eq, are left-linear, i.e., all the variables within clause heads occur only once.

Roughly speaking, FLPs are constructed using a set of basic combinators which are expressive enough to get the full power of logic programming. First, the eq predicate allows us to simulate structure sharing within terms, which is usually achieved using the same variable names in clause heads; secondly, cases (i) and (ii) allow us to make use of the logic connectives \( \land \) and \( \lor \), denoted, as usual in logic programming, by \( ",\" \) and \( \";\" \); thirdly, case (iii) allows the introduction of local variables within clause bodies; and finally, case (iv) allows us to exploit unification and value passing by means of nontrivial terms within clause heads and bodies. Obviously, each logic program can be turned into an FLP by a semantics-preserving transformation. The following example illustrates the transformation process.
Example 4.1. The following program for appending lists:

\[
\text{append}(\text{nil}, x, x) : -
\]
\[
\text{append}(\text{cons}(x, y), z, \text{cons}(x, w)) : - \text{append}(y, z, w)
\]
can be factorized as follows:

\[
\text{append}(x, y, z) : - \text{appendl}(x, y, z)
\]
\[
\text{append}(x, y, z) : - \text{append2}(x, y, z)
\]
\[
\text{appendl}(\text{nil}, y, z) : - \text{eq}(y, z)
\]
\[
\text{append2}(\text{cons}(x, y), z, \text{cons}(u, w)) : - \text{eq}(x, u)
\]
\[
\text{append3}(\text{cons}(x, y), z, \text{cons}(v, w)) : - \text{eq}(x, v)
\]
\[
\text{append4}(\text{cons}(x, y), z, \text{cons}(v, w)) : - \text{eq}(x, z)
\]

It is obvious, although tedious, to verify that the semantics of the original predicate append is preserved.

In view of all this it is possible to develop the formalization of the transformation technique only with respect to FLPs, without lack of generality. The aim of such a transformation is to synthesize the definitions of a new predicate \( \bar{p} \) for each predicate \( p \) in an FLP \( P \). The intended meaning of \( \bar{p} \) is the negation of the original one. As is well known, negative information can be obtained from a logic program only by referring to the completed program itself (the completed program associated to \( P \) is called the completion of \( P \) in [15]). Hence, the transformation is applied to the completed program \( P^* \) of an FLP \( P \), which is given in the following definitions.

Definition 4.2. Given an FLP \( P \), the completed definition of a predicate occurring in it is defined as follows:

(a) The completed definition of eq is

\[
\forall xy (eq(x, y) \leftrightarrow x = y).
\]

(b) The completed definition of a predicate defined as in Definition 4.1(i), (ii), (iii), and (iv) is respectively:

(i) \( \forall \bar{x} (p(\bar{x}) \leftrightarrow q(\bar{x}), r(\bar{x})) \),

(ii) \( \forall \bar{x} (p(\bar{x}) \leftrightarrow q(\bar{x}) \lor r(\bar{x})) \),

(iii) \( \forall \bar{x} (p(\bar{x}) \leftrightarrow \exists y q(\bar{x}, y)) \),

(iv) \( \forall \bar{y} (p(\bar{y}) \leftrightarrow \exists \bar{x} (\bar{x} = \bar{i}, q(\bar{i}))) \),

where \( \bar{x} \) are the variables occurring in \( i \) and \( \bar{y} \) are fresh variables.

(c) The completed definition of a predicate \( p \) occurring in \( P \), but in no clause head, is

\[
\forall \bar{x} (p(\bar{x}) \leftrightarrow \text{false}).
\]

Definition 4.3. The completed program \( (P^*, U^*_{DCA}) \) associated to an FLP \( P \) is the collection of:

(i) the completed definition of each predicate in \( P \) as in Definition 4.2,

(ii) the axioms \( U^*_{DCA} \) augmented with the axioms \( \forall \bar{x} \bar{z} (\bar{x} = \bar{z} \rightarrow p(\bar{x} = p(\bar{z})) \) for each predicate \( p \) in \( P \).
We are now in the position to show how the technique works and how it is achieved by means of a step-by-step transformation of the completed definitions in $(P^*, U_{DCA})$.

First of all, consider the logical negation of both sides of each formula in Definition 4.2(b), thus preserving the original meaning:

(i) $\forall \bar{x}(-p(\bar{x}) \leftrightarrow \neg q(\bar{x}) \lor \neg r(\bar{x}))$,

(ii) $\forall \bar{x}(-p(\bar{x}) \leftrightarrow \neg q(\bar{x}), \neg r(\bar{x}))$,

(iii) $\forall \bar{x}(-p(\bar{x}) \leftrightarrow \forall \bar{y} \neg q(\bar{x}, \bar{y}))$,

(iv) $\forall \bar{y}(-p(\bar{y}) \leftrightarrow \forall \bar{x}(\bar{y} \neq \bar{i} \lor \neg q(i')))$. 

Secondly, notice that case (iv) above can be trivially transformed into

$\forall \bar{y}(-p(\bar{y}) \leftrightarrow \forall \bar{x}. \bar{y} \neq \bar{i} \lor \exists \bar{x}(\bar{y} = \bar{i}, \neg q(i')))$, 

still preserving the logical meaning. Finally, as we are working under the theory $(P^*, U_{DCA})$, by Theorem 3.1 the formula $\forall \bar{x}. \bar{y} \neq \bar{i}$ can be replaced by

$\exists \bar{x}_1. \bar{y} = \bar{i}_1 \lor \cdots \lor \exists \bar{x}_m. \bar{y} = \bar{i}_m$, 

where $\{\bar{i}_1, \ldots, \bar{i}_m\} = \text{Not}(\bar{i})$. Thus we further transform the formula into the equivalent one

$\forall \bar{y}(-p(\bar{y}) \leftrightarrow \exists \bar{x}_1. \bar{y} = \bar{i}_1 \lor \cdots \lor \exists \bar{x}_m. \bar{y} = \bar{i}_m \lor \exists \bar{x}(\bar{y} = \bar{i}, \neg q(i')))$. 

or, in a more compact notation,

(iv') $\forall \bar{y}(-p(\bar{y}) \leftrightarrow \forall i=1, \ldots, m(\exists \bar{x}_i. \bar{y} = \bar{i}_i) \lor \exists \bar{x}(\bar{y} = \bar{i}, \neg q(i')))$. 

Up to now, we have obtained an equivalent representation of $(P^*, U_{DCA})$ in terms of negative literals. The idea now is to use a new predicate symbol $\bar{p}$ wherever a negative literal $-p(\bar{u})$ occurs, for each predicate symbol $p$. In this way the formulae (i), (ii), (iii) and (iv') above are rewritten as

(i'') $\forall \bar{x}(\bar{p}(\bar{x}) \leftrightarrow \bar{q}(\bar{x}) \lor \bar{r}(\bar{x}))$,

(ii'') $\forall \bar{x}(\bar{p}(\bar{x}) \leftrightarrow \bar{q}(\bar{x}), \bar{r}(\bar{x}))$,

(iii'') $\forall \bar{x}(\bar{p}(\bar{x}) \leftrightarrow \forall \bar{y}. \bar{q}(\bar{x}, \bar{y}))$,

(iv'') $\forall \bar{y}(\bar{p}(\bar{y}) \leftrightarrow \forall i=1, \ldots, m(\exists \bar{x}_i. \bar{y} = \bar{i}_i) \lor \exists \bar{x}(\bar{y} = \bar{t}, \bar{q}(\bar{i}')))$. 

Consider now the logical negation of both sides of a formula as in Definition 4.2(c), yielding

$\forall \bar{x}(-p(\bar{x}) \leftrightarrow \text{true})$. 

Again, if a new predicate symbol $\bar{p}$ is used where $-p$ occurs, we get

$\forall \bar{x}(\bar{p}(\bar{x}) \leftrightarrow \text{true})$. 

It is important to stress that, apart from the occurrences of the new predicates $\bar{p}$ where originally a negative literal occurred, all these formulae, under the theory $(P^*, U_{DCA})$, are equivalent to the original ones occurring in the completed program.

The next step is to consider these formulae as the completed definitions of the new predicate symbols $\bar{p}$. This can be done easily in some of the above cases, as
shown here:

\[ \forall x \ ((\bar{p}(x) \iff \text{true}) \quad \text{gives the single fact} \quad \bar{p}(x) : \neg) \]
\[ \forall x \ ((\bar{p}(x) \iff \bar{q}(x) \lor \bar{r}(x)) \quad \text{gives two rules} \quad \bar{p}(x) : \neg \bar{q}(x); \bar{r}(x)) \]
\[ \forall x \ ((\bar{p}(x) \iff \bar{q}(x), \bar{r}(x)) \quad \text{gives the single rule} \quad \bar{p}(x) : \neg \bar{q}(x), \bar{r}(x)) \]
\[ \forall y ((\bar{p}(y) \iff \forall_{i_1, \ldots, i_m} (\exists x. \bar{y} = i_j) \lor \exists x (y = i, \bar{q}(i'))) \quad \text{gives} \ m \ \text{facts} \quad \bar{p}(i_j) : \neg \]
and the rule \( \bar{p}(i') : \neg \bar{q}(i') \).

The major problem arises in finding the appropriate definition of \( \bar{p} \) when local
variables occur in the right-hand side of its definition as in (iii"). Indeed, those
formulae cannot be turned into clauses, since universally quantified literals would
appear in their right-hand sides. This is due to the fact that the local variables,
which were existentially quantified in the original program, are now universally
quantified. In other words, the formula

\[ \forall x ((\bar{p}(x) \iff \forall y. \bar{q}(x, y)) \]

would result into the following definition for \( \bar{p} \):

\[ \bar{p}(x) : \neg \forall y. \bar{q}(x, y) \]

Nevertheless, a simple way to implement a weak form of universal quantification
can be devised using our technique combined with negation as failure.

A subgoal as \( \forall y. \bar{q}(x, y) \), where \( y \) are the local variables, can be computed in two
steps:

(1) Evaluate the subgoal \( \bar{q}(x, y) \), using the clauses for the \( \bar{q} \) predicate. Let \( \theta \) be
a computed answer substitution for such a goal.

(2) Evaluate the subgoal \( q(x, z) \theta \), where \( z \) is a tuple of new variables which
replace the original local variables \( y \). If such an evaluation is \textit{finitely failed},
then \( \theta \) is a computed answer substitution for the whole subgoal \( \forall y. \bar{q}(x, y) \).

Roughly speaking, the universal quantification is computed in a kind of generate-
and-test style. Indeed, the evaluation of the subgoal \( \bar{q}(x, y) \) generates candidate
substitutions for \( x \), whereas the evaluation of the subgoal \( q(x, z) \theta \), where the
computed values for the local variables have been disregarded, checks, under
negation as failure, whether such values actually make the original predicate finitely
fail. In this way, negation as failure is used to select the correct results, since it
checks universal negative theorems. Actually, this solution does not guarantee
completeness of the query evaluation process as it will be pointed out in the next
section. Nevertheless, as will be discussed in detail below, this solution fails to work
properly only in very peculiar cases.

In order to complete the transformation, the following definition is added:

\[ \forall x ((\bar{p}(x) \iff \forall y. \bar{q}(x, y)) \quad \text{gives the rule} \quad \bar{p}(x) : \forall y. \bar{q}(x, y) \]

where \( \forall y. \bar{q}(x, y) \) is actually shorthand for the construction

\[ \langle \bar{q}(x, y), \textit{naf}_q(x, z) \rangle \]

term a \textit{naf subgoal}. The notation \( \textit{naf}_q(x, z) \) stresses the fact that possible
bindings for the local variables are disregarded, since \( \bar{z} \) are fresh variables. 
\( \text{naf}_q(\bar{x}, \bar{z}) \) is termed a \textit{naf literal}.

Finally, the transformation for the eq predicate, which is defined by the single clause \( \text{eq}(x, x) :- \), is needed. The clause for \( \text{eq} \) is not left-linear; hence an ad hoc transformation has to be provided. In fact, the rules for \( \text{eq} \) are parametrized with respect to the underlying language \( L \), as follows:

\[
\text{eq}(c_1(\bar{x}), c_j(\bar{y})) :-
\]
for each pair of different constructor symbols \( c_i \) and \( c_j \),

\[
\text{eq}(c_i(x_1, \ldots, x_k), c_j(y_1, \ldots, y_k)) :- \text{eq}(x_j, y_j)
\]
for each constructor symbol \( c_i \), for each argument position \( j \in [1, k_i] \).

Notice that the above rules actually constitute a standard logic program, and it is easy to see that its minimal model is the complement of the identity relation over the Herbrand universe \( H_L \). It is worth mentioning that the definition of \( \text{eq} \) has some disadvantages. For instance \( \text{eq}(x, f(x)) \), which intuitively should succeed without binding \( x \), produces an infinite set of answer substitutions. Some of these problems can be solved if subgoals of the kind \( \text{eq}(s, t) \) are treated as constraints similar to those in CLP [12].

\textbf{Example 4.2.} Given the language introducing constants \( a, b \) and the binary function \( f \), \( \text{eq} \) is defined as follows.

\[
\text{eq}(a, b) :-
\text{eq}(a, f(x, y)) :-
\text{eq}(b, a) :-
\text{eq}(b, f(x, y)) :-
\text{eq}(f(x, y), a) :-
\text{eq}(f(x, y), b) :-
\text{eq}(f(x, y), f(v, w)) :- \text{eq}(x, v)
\text{eq}(f(x, y), f(v, w)) :- \text{eq}(y, w).
\]

The transformation technique for FLPs is summarized in the following definition of the negated program \( P_{\text{neg}} \).

\textbf{Definition 4.4.} Given an FLP \( P \), \( P_{\text{neg}} \) is defined as follows:

(a) For predicates defined as in Definition 4.1(i), (ii), (iii), and (iv) it contains the definitions

(i) \( \tilde{p}(\bar{x}) :- \tilde{q}(\bar{x}); \tilde{r}(\bar{x}) \),
(ii) \( \tilde{p}(\bar{x}) :- \tilde{q}(\bar{x}); \tilde{r}(\bar{x}) \),
(iii) \( \tilde{p}(\bar{x}) :- \forall \bar{y}. \tilde{q}(\bar{x}, \bar{y}) \),
(iv) \( \tilde{p}(i_1) :- \)
  \( \quad \vdots \)
  \( \quad \tilde{p}(i_m) :- \)
  \( \quad \tilde{p}(i) :- \tilde{q}(i') \),
where \( \{i_1, \ldots, i_m\} = \text{Not}(i) \).
(b) For the predicate \( \widetilde{eq} \) it contains the definitions
\[
\widetilde{eq}(c_i(x), c_j(y)) : -
\]
for each pair of different constructor symbols \( c_i \) and \( c_j \),
\[
\widetilde{eq}(c_i(x_1, \ldots, x_k), c_i(y_1, \ldots, y_k)) : - \widetilde{eq}(x_j, y_j)
\]
for each constructor symbol \( c_i \), for each argument position \( j \in [1, k] \).

(c) For each predicate symbol \( p \) occurring in some bodies of clauses of \( P \) but in no clause head it contains the definition
\[
\bar{p}(\bar{x}) : -
\]

5. THE QUERY EVALUATION PROCESS

Once the negated program \( P_{\text{neg}} \) has been obtained from an FLP \( P \), a suitable refutation procedure has to be defined in order to extract negative information from it. Such a refutation procedure, called SLDN, is essentially SLD resolution except that it has to cope with naf subgoals within rule bodies. This is the case when a rule contains a conjunction of the kind
\[
\langle \bar{q}(\bar{x}, \bar{y}), \text{naf}_{\text{s}}q(\bar{x}, \bar{z}) \rangle,
\]
which is the result of transforming program clauses with local variables [recall that the local variables in \( \text{naf}_{\text{s}}q(\bar{x}, \bar{z}) \) are different from the ones in \( \bar{q}(\bar{x}, \bar{y}) \)]. As mentioned in the previous section, the evaluation of such a conjunction should be carried out in a generate-and-test fashion, namely, \( \bar{q}(\bar{x}, \bar{y}) \) should be evaluated first, in order to provide candidate solutions for \( \bar{x} \), whereas \( \text{naf}_{\text{s}}q(\bar{x}, \bar{z}) \) should act as a filter, in order to disregard the incorrect solutions. Hence, the evaluation of such a conjunction has to be carried out from left to right, that is, negation as failure should be invoked only when the evaluation of \( \bar{q}(\bar{x}, \bar{y}) \) is completed, i.e. when the naf subgoal is in the form \( \langle \emptyset, \text{naf}_{\text{s}}q(\bar{x}, \bar{z}) \rangle \). Although at first sight the use of negation as failure when evaluating naf literals seems sufficient to get the desired behavior, this is not the case. Referring again to \((\ast)\), the problem is that all the candidate substitutions for \( \bar{x} \) provided by \( \bar{q}(\bar{x}, \bar{y}) \) can make the evaluation of \( q(\bar{x}, \bar{z}) \) succeed [and hence the evaluation of \( \text{naf}_{\text{s}}q(\bar{x}, \bar{z}) \) fail], even if there exists a substitution \( \theta \) for \( \bar{x} \) such that \( \forall \bar{y} \neg q(\bar{x}, \bar{y}) \). Since the major aim of using \( P_{\text{neg}} \) is to be able to compute negative information, this all would result in an incomplete query evaluation process. Here is an example of such an undesired behavior which can help in understanding the problem as well as providing hints for the solution.

**Example 5.1.** Consider the following logic program over a language with two constants \( a, b \):
\[
p(x, y, z) : - q(x, y, z), r(x, y, z)
q(b, x, y) : -
q(x, y, b) : -
r(x, y, a) : -
r(x, b, y) : -
s(x, y) : - p(x, y, z).
\]
First of all, let us take the FLP $P$ corresponding to it:

$$p(x, y, z) :- q(x, y, z), r(x, y, z)$$
$$q(x, y, z) :- q_1(x, y, z), q_2(x, y, z)$$
$$q_1(b, x, y) :-$$
$$q_2(x, y, b) :-$$
$$r(x, y, z) :- r_1(x, y, z), r_2(x, y, z)$$
$$r_1(x, y, a) :-$$
$$r_2(x, b, y) :-$$
$$s(x, y) :- p(x, y, z).$$

The negated program $P_{neg}$ computed by the transformation technique is

$$\bar{p}(x, y, z) :- \bar{q}(x, y, z), \bar{r}(x, y, z)$$
$$\bar{q}(x, y, z) :- \bar{q}_1(x, y, z), \bar{q}_2(x, y, z)$$
$$\bar{q}_1(a, x, y) :-$$
$$\bar{q}_2(x, y, a) :-$$
$$\bar{r}_1(x, y, z) :- \bar{r}_1(x, y, z), \bar{r}_2(x, y, z)$$
$$\bar{r}_1(x, y, b) :-$$
$$\bar{r}_2(x, a, y) :-$$
$$\bar{s}(x, y) :- \bar{p}(x, y, z), \text{naf} p(x, y, w).$$

It is easy to see that the ground atom $s(a, a)$ belongs to the finite failure set of $P$. Hence, the SLDN evaluation of, say, $\bar{s}(x, y)$ should yield an answer substitution having $s(a, a)$ as a ground instance. Let us follow the major evaluation steps performed by the refutation procedure, starting from $\bar{s}(x, y)$:

1. Evaluate the conjunction $\bar{p}(x, y, z), \text{naf}_w p(x, y, w)$.

2. One refutation of $\bar{p}(x, y, z)$ gives $\{x = a, y = \_, z = a\}$ as answer substitution, but the evaluation of $\text{naf}_w p(a, \_, w)$ fails, since $p(a, \_, w)$ has a successful SLD derivation.

3. The second refutation of $\bar{p}(x, y, z)$ gives $\{x = \_, y = a, z = b\}$ as answer substitution, but again the evaluation of $\text{naf}_w p(\_, a, w)$ fails, since $p(\_, a, w)$ has a successful derivation.

The point here is that the failure of $s(a, a)$ is due, ultimately, to the failure of the conjunction $q(x, y, z), r(x, y, z)$, but neither $q(a, a, z)$ alone nor $r(a, a, z)$ alone fails. On the other hand, the candidate substitutions produced during the evaluation of $\bar{s}(x, y)$ are computed separately by $\bar{q}(x, y, z)$ and $\bar{r}(x, y, z)$: they both bind the variable $z$, but such a binding is disregarded once negation as failure is invoked. The final result is that the candidate substitutions are not specialized enough to make the corresponding naf literal succeed. This example, although simple, shows how local variables are the real source of trouble for the transformation technique, and in particular their interaction with conjunction. In other words, when candidate solutions for naf literals are produced separately by literals corresponding to a
conjunction in the original program, the bindings produced that way can be too
general to get successful derivations.

Nevertheless, the example itself suggests a solution which can be confined within
the evaluation of naf literals. Consider for instance step (2) above: once the
successful derivation of \( p(a, _, w) \) has been discovered, one should require that
further instantiations of the nonlocal variables of \( \text{naf}_w p(a, _, w) \) make the naf literal
itself succeed. In this case, since the underlying language provides only the constants
\( a \) and \( b \) and no function, two instantiations are possible, namely \( \text{naf}_w p(a, a, w) \)
and \( \text{naf}_w p(a, b, w) \), and, in fact, the first one succeeds, since \( p(a, a, w) \) has only
finitely failed SLD derivations. Thus, the whole computed substitution for the
evaluation of \( s(x, y) \) would be \( \{ x = a, \ y = a \} \), as expected.

In general, the evaluation of the naf literal \( \text{naf}_t p(t, \bar{z}) \) (where \( t \) is the current
candidate solution and \( \bar{z} \) are the local variables) should be carried out as follows:

1. If \( p(i, \bar{z}) \) has only finitely failed SLD derivations, then succeed with no
   further instantiation.
2. Otherwise, instantiate the variables in \( i \), obtaining \( i' \), and repeat from step
   (1).

It remains to show how to achieve the instantiations required in step (2). Referring to the underlying language \( L \), this can be done by means of a predicate, say herbrand, generating all possible Herbrand terms. The definition of herbrand could be the following:

\[
\text{herbrand}(x) :-
\text{herbrand}(c_i(x_1, \ldots, x_k)) :- \text{herbrand}(x_1), \ldots, \text{herbrand}(x_k)
\]

for each constructor symbol \( c_i \).

The second clause schema reduces to a fact schema when the arity of \( c_i \) is 0, i.e., it is
a constant.

\textbf{Example 5.2.} Referring to the language of Example 4.2, herbrand is defined as
follows:

\[
\text{herbrand}(x) :-
\text{herbrand}(a) :-
\text{herbrand}(b) :-
\text{herbrand}(f(x, y)) :- \text{herbrand}(x), \text{herbrand}(y).
\]

Thus, \( \text{naf}_t q(\bar{x}, \bar{z}) \) can be turned into a conjunction of the kind

\[
(\text{herbrand}(\bar{x}), \text{naf}_t q(\bar{x}, \bar{z})),
\]

which again has to be evaluated from left to right [the meaning of \( \text{herbrand}(\bar{x}) \)
should be obvious]. Hereafter, as far as the formal treatment is concerned, a naf
subgoal will be understood as the above conjunction, embedding the herbrand
predicate. The following definitions formalize SLDN resolution.
Definition 5.1. A (SLDN) goal $G$ is recursively defined as follows:

(i) $\emptyset$ (empty goal);
(ii) $\bar{p}(\bar{i})$ (ordinary literal);
(iii) $(G', \text{naf } \bar{z} p(\bar{i}, \bar{z}))$ (naf subgoal), where $G'$ is an (SLDN) goal;
(iv) $G_1, \ldots, G_n$ (conjunction), where the $G_i$'s are (SLDN) goals.

Definition 5.2. A computation rule $R$ for SLDN resolution is such that, given a goal, either a literal or a naf subgoal is selected for computation.

This definition summarizes the intuitive considerations stated above, namely that naf literals have to be evaluated only when candidate solutions have been provided.

Definition 5.3. Given a goal $G_k = L_1, \ldots, L_n$ and a computation rule $R$ for SLDN, let $L_i$ be selected by $R$. Then $G_{k+1}$ is SLDN-derived from $G_k$ via $\theta_{k+1}$ if one of the following conditions holds:

(i) $L_i = \bar{p}(\bar{i}) [L_i = \text{eq}(t, u)]: \theta_{k+1}$ and $G_{k+1}$ are obtained through an ordinary SLD-resolution step, using the clauses for $\bar{p}$ [eq];
(ii) $L_i = (G', \text{naf } \bar{z} p(\bar{i}, \bar{z}))$: $G_{k+1} = (L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n)\theta_{k+1}$ if $\theta_{k+1}$ is a solution of herbrand($i$), and $p(i\theta_{k+1}, \bar{z})$ has only finitely failed SLD derivations;
(iii) $L_i = (G', \text{naf } \bar{z} p(\bar{i}, \bar{z}))$: $G_{k+1} = (L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_n)\theta_{k+1}$, where $L_i = (G'', \text{naf } \bar{z} p(\bar{i}, \bar{z}))$, $G''$ is SLDN-derived from $G'$ via $\theta_{k+1}$.

Some remarks about the above definition are in order. First of all, there is a predefined evaluation strategy for the naf subgoals. During a computation, naf subgoals are introduced in the form $(\bar{q}(\bar{i}, \bar{y}), \text{naf } \bar{z} q(\bar{i}, \bar{z}))$; as a consequence an SLDN refutation of $(\bar{q}(\bar{i}, \bar{y}), \text{naf } \bar{z} q(\bar{i}, \bar{z}))$ has to be completed in order to provide a candidate substitution $\theta$, before evaluating $\text{naf } \bar{z} q(i\theta, \bar{z})$. This behavior is reflected in cases (ii) and (iii) of the previous definition. Secondly, it is worth noting that negation as failure is used here only as a means to check universal-quantified statements, and for this reason SLDN resolution, unlike SLDNF resolution [15], never flounders.

Definition 5.4. Let $R$ be a computation rule for SLDN resolution. An SLDN derivation of the goal $G$ is a sequence of goals $G_0 = G, G_1, G_2, \ldots$ such that $G_{i+1}$ is SLDN-derived from $G_i$ via $\theta_{i+1}$. An SLDN refutation of $G$ is a finite SLDN derivation $G_0 = G, G_1, \ldots, G_n$ such that $G_n = \emptyset$. The restriction of the substitution $\theta_1, \ldots, \theta_n$ to the variables of $G$ is called a computed answer substitution. A finite SLDN derivation of $G$ which does not end with $\emptyset$ is said to be failed.

As in the case of standard SLD resolution, it is easy to prove the independence of SLDN resolution from the computation rule, provided that it actually selects either a literal of a whole naf subgoal. The only difference between SLD and SLDN resolution is in the evaluation of subgoals such as $(\emptyset, \text{naf } \bar{z} p(\bar{i}, \bar{z}))$, but even in this
case the independence property is guaranteed by the use of the herbrand predicate before invoking negation as failure. The following statement formalizes the property; its proof is only sketched, since it is a simple extension of the proof for SLD resolution given in [15].

**Lemma 5.1.** Let $P_{neg}$ be the transformation of an FLP $P$, $R$, a computation rule for SLDN resolution, and $G$ a goal. Suppose that there exists an SLDN refutation $G_0 = G, G_1, \ldots, G_n = \emptyset$ of $G$ using $R$ which yields an answer substitution $\theta$. Suppose that $L_i$ is selected by $R$ in $G_k$, and $L_j$ is selected by $R$ in $G_{k+1}$. Then there exists an SLDN refutation of $G$ using $R'$ which is the same as $R$ except that $L_j$ is selected in $G_k$, and $L_i$ is selected from $G_{k+1}$. Moreover, if $R'$ yields $\theta'$ as answer substitution, then $G\theta$ is equal to $G\theta'$ up to variable renaming.

**Proof.** If $L_i$ and $L_j$ are ordinary literals or naf subgoals of the kind $\langle G, \text{naf}, p(i, z) \rangle$ with $G \neq \emptyset$, then the proof is similar to the one of Lemma 9.1 in [15]. Suppose that $L_i$ is a naf subgoal of the kind $\langle 0, \text{naf}, p(i, z) \rangle$. Then $\theta_k$ is a solution of herbrand such that $p(i\theta_k, z)$ has only finitely failed SLD derivations. Now, suppose that $L_j$ is selected in $G_k$ instead of $L_i$ and that the next goal is obtained via the substitution $\theta'_k$. The naf subgoal $\langle 0, \text{naf}, p(i\theta'_k, z) \rangle$ is evaluated next, and, by definition of herbrand, there exists a solution of herbrand($i\theta'_k$), say $\theta'_{k+1}$, such that $\theta'_k \theta'_{k+1}$ is equal to $\theta_k \theta'_{k+1}$ up to variable renaming. The refutation of the rest of the goal using $R'$ can now proceed like the corresponding one using $R$; hence the thesis. The remaining cases can be proved in a similar way. $\square$

Repeated applications of the above lemma give a simple proof of the following

**Proposition 5.1.** Let $P_{neg}$ be the transformation of an FLP $P$, $G$ a goal, and $R, R'$ two computation rules for SLDN-resolution. If $G$ has an SLDN refutation using $R$ which yields $\theta$, then $G$ has also an SLDN refutation using $R'$ which yields $\theta'$, such that $G\theta$ and $G\theta'$ are equal up to variable renaming.

Proposition 5.1 enables us to devise a class of equivalent computation rules which are easy to implement, namely those which select ordinary literals as far as possible. This means that the invocation of negation as failure is delayed until nothing else can be performed. In this way, candidate substitutions are as specialized as possible without invoking the herbrand predicate, so that there is a greater chance that the negation-as-failure checks succeed without further instantiations. Our experience with the implemented system showed that this is actually the case in most situations.

6. **SOUNDNESS AND COMPLETENESS OF SLDN RESOLUTION**

Soundness and completeness of SLDN resolution are stated here with respect to the completion $(P^*, U^*_DCA)$, which constitute the referring theory for the transformation technique. Indeed, both results are corollaries of analogous results which hold with respect to the standard completion $(P^*, U^*)$. The link is given by the relationships between $(P^*, U^*)$ and $(P^*, U^*_DCA)$ with respect to universally quantified negative theorems, which are discussed in [18] and can be summarized in the following
Proposition 6.1. Let $P$ be a logic program and $p(i)$ be an atom. The following statements are equivalent:

(i) $(P^*, U^*) \vdash \forall \neg p(i)$.

(ii) $(P^*, U^*) \vdash \forall \neg p(i)$.

(iii) $p(i)$ has only finitely failed fair SLD derivations.

Notice that the equivalence between (ii) and (iii) summarizes the standard soundness and completeness results of fair SLD resolution with respect to standard completion, while the equivalence between (i) and (iii) gives a stronger notion of completeness for negation as failure, exploiting the equivalence of $(P^*, U^*)$ and $(P^*, U_{DCA}^*)$ (which is actually a stronger theory) with respect to negative information.

The soundness result for SLDN resolution relates computed answer substitutions to universally quantified negative theorems of the completed program $(P^*, U^*)$. In order to simplify the proof, we refer to SLDN resolution using a computation rule of the class mentioned at the end of the previous section. This enables us to look at a goal as a conjunction of the kind

$$\bar{p}_1(\tilde{s}_1), \ldots, \bar{p}_k(\tilde{s}_k), \text{naf}_{\tilde{s}_1} p_{k+1}(\tilde{s}_{k+1}, \tilde{z}_1), \ldots, \text{naf}_{\tilde{s}_m} p_{k+m}(\tilde{t}_{k+m}, \tilde{z}_m)$$

and to understand the computation rule as selecting a subgoal of the kind $\text{naf}_i p(i, \bar{z})$ only if no ordinary literal occurs in the goal. Finally, given a goal $G$ as above and a substitution $\theta$ for the free variables of $G$, we will denote by the pair $(-G, \theta)$ the formula

$$\forall(\neg p_1(\tilde{s}_1), \ldots, \neg p_k(\tilde{s}_k), \neg p_{k+1}(\tilde{s}_{k+1}, \tilde{z}_1), \ldots, \neg p_{k+m}(\tilde{t}_{k+m}, \tilde{z}_m))\theta.$$

Theorem 6.1. Let $P$ be an FLP, $P_{neg}$ its transformation, and $G$ a goal. Then each computed answer substitution $\theta$ for $G$ is such that

$$(P^*, U^*) \vdash (-G, \theta).$$

PROOF. The proof is carried out by induction on the length of the SLDN derivation yielding $\theta$.

Base step: The following cases are possible:

(i) $G = \bar{p}_1(\tilde{s}_1)$, and the empty goal is derived from $G$ using the clause

$$\bar{p}_1(i) : -.$$

This means that the original program $P$ contains for $p_1$ only one clause of the kind

$$p_1(i) : q(i')$$

such that $\bar{i'} \in \text{Not}(i)$. Moreover the computed answer substitution is $\theta$-mgu$(\tilde{s}_1, \bar{i})$, and $\bar{i', \theta}$ does not unify with $i$. This implies that $p_1(i, \theta)$ does not match any clause in $P$, i.e., $(P^*, U^*) \vdash (-G, \theta)$ by the standard soundness result for SLD resolution.

(ii) $G = \bar{p}_1(\tilde{x})$, and the empty goal is derived from $G$ using the clause

$$\bar{p}_1(\bar{x}) : -.$$
This means that the original program $P$ does not contain any clause for $p_1$, and the thesis holds trivially.

(iii) $G = \text{eq}(t_1, t_2)$, and the empty goal is derived from $G$ using the clauses for $\text{eq}$. Since the minimal model of the logic program defining $\text{eq}$ is the complement of the identity relation over $H_L$, we have immediately the thesis.

(iv) $G = \text{naf} \neg p_1(i_1, z_1)$. By definition of SLDN, the computed answer substitution $\theta$ is a solution of the goal $\text{herbrand}(i_1)$ such that $p_1(i_1\theta, z_1)$ has only finitely failed SLD derivations. By the standard soundness result on the negation-as-failure rule, we have $(P^*, U^*) \models \forall \neg p_1(i_1\theta, z_1)$, i.e. the thesis.

**Inductive Step:** Assume that the thesis holds for answer substitutions computed by SLDN refutations of length less or equal than $n - 1$. The following cases arise:

1. $G = (\text{naf} \neg p_1(i_1, z_1), G')$, and $\text{naf} \neg p_1(i_1, z_1)$ is selected for the evaluation (recall that this means that no ordinary literal occurs in the goal). The derived goal is $G'\theta'$, where $\theta'$ is a solution of the goal $\text{herbrand}(i_1)$ such that $p_1(i_1\theta', z_1)$ has only finitely failed SLD derivations. The inductive hypothesis states that
   
   $$(P^*, U^*) \models \neg G'\theta', \theta''$$
   
   and the whole computed answer substitution is $\theta'\theta''$. This last observation, along with the fact that
   
   $$(P^*, U^*) \models \forall \neg p_1(i_1\theta', z_1)$$
   
   implies $$(P^*, U^*) \models \forall \neg p_1(i_1\theta'\theta'', z_1),$$
   
   gives the thesis.

In the rest of the proof, let us assume, without loss of generality, that the goal is $\text{p}_1(\tilde{s}_1), G'$ and the selected literal is $\text{p}_1(\tilde{s}_1)$. The proof goes by case analysis on the rule used in the first SLDN-resolution step.

2. $P_{\neg \text{eq}}$ contains the following definition of $\text{p}_1$:
   
   $$\text{p}_1(\bar{x}) := \neg q(\bar{x}); r(\bar{x}),$$
   
   and the derived goal is $G_1 = (\text{eq}(\tilde{s}_1), G')$. Notice that the substitution computed in this step does not apply to the variables in $G'$. By the definition of the transformation, $(P^*, U^*)$ contains the axiom
   
   $$\forall \bar{x} (p_1(\bar{x}) \leftrightarrow q(\bar{x}), r(\bar{x})).$$
   
   This, along with the fact that the inductive hypothesis $(P^*, U^*) \models \neg G_1, \theta$ implies
   
   $$(P^*, U^*) \models \forall \neg q(\tilde{s}_1\theta),$$
   
   gives
   
   $$(P^*, U^*) \models \forall \neg q(\tilde{s}_1\theta), \neg G', \theta$$
   
   implies $(P^*, U^*) \models \forall \neg p_1(\tilde{s}_1\theta), \neg G', \theta$;
   
   hence
   
   $$(P^*, U^*) \models \neg G, \theta.$$
(3) \( P_{\text{neg}} \) contains the following definition of \( \bar{p}_1 \):
\[
\bar{p}_1(\bar{x}) := \neg q(\bar{x}), \neg r(\bar{x}),
\]
and the derived goal is \( G_1 = (\bar{q}(\bar{s}_1), \bar{r}(\bar{s}_1), G') \). Notice that, again, the substitution computed in this step does not apply to the variables in \( G' \). By the definition of the transformation, \((P^*, U^*)\) contains the axiom
\[
\forall \bar{x}(p_1(\bar{x}) \leftrightarrow q(\bar{x}) \lor r(\bar{x})).
\]
This, along with the fact that the inductive hypothesis \((P^*, U^*) \vdash (\neg G_1, \theta)\) implies
\[
(P^*, U^*) \vdash \forall (\neg q(\bar{s}, \theta), \neg r(\bar{s}, \theta)),
\]
gives
\[
(P^*, U^*) \vdash \forall (\neg q(\bar{s}, \theta), \neg r(\bar{s}, \theta)) \land (\neg G', \theta)
\]
implies \((P^*, U^*) \vdash \forall \neg p_1(\bar{s}, \theta), (\neg G', \theta)\);
hence
\[
(P^*, U^*) \vdash (\neg G, \theta).
\]

(4) \( P_{\text{neg}} \) contains the following definition of \( \bar{p}_1 \):
\[
\bar{p}_1(\bar{x}) := \langle q(\bar{x}, \bar{y}), \text{naf } q(\bar{x}, \bar{z}) \rangle,
\]
and the derived goal is \( G_1 = (\bar{q}(\bar{s}_1, \bar{y}), \text{naf } q(\bar{s}_1, \bar{z}), G') \). Notice that, again, the substitution computed in this step does not apply to the variables in \( G' \). The inductive hypothesis
\[
(P^*, U^*) \vdash (\neg G_1, \theta)
\]
implies the following two facts:
(i) \((P^*, U^*) \vdash \forall \neg q(\bar{s}, \theta, \bar{z})\),
(ii) \((P^*, U^*) \vdash (\neg G', \theta)\).

Since \((P^*, U^*)\) contains the axiom \( \forall \bar{x}(p_1(\bar{x}) \leftrightarrow \exists \bar{y}. q(\bar{x}, \bar{y})) \), fact (i) above implies that \((P^*, U^*) \vdash \forall \neg p_1(\bar{s}, \theta)\). This, along with (ii), gives
\[
(P^*, U^*) \vdash \forall \neg p_1(\bar{s}, \theta), (\neg G', \theta);
\]
hence
\[
(P^*, U^*) \vdash (\neg G, \theta).
\]

(5) \( P_{\text{neg}} \) contains the following definition of \( \bar{p}_1 \):
\[
\bar{p}_1(i_1) := \cdots \bar{p}_1(i_m) := \bar{p}_1(\bar{i}) := \neg q(\bar{i}).
\]
Then two cases arise. First, the derived goal is \( G_1 = G' \theta_i \), with \( \theta_i = \text{mgu}(\bar{s}_1, \bar{i}) \), i.e., it is obtained using one of the \( m \) facts above. Recall that \( i \in \text{Not}(i) \), which implies that \( \bar{s}_1, i \) does not unify with \( i \). Since the only clause for \( p_1 \) in \( P \) is
\[
p_1(\bar{i}) := q(\bar{i}),
\]
by the soundness of SLD resolution we have
\[
(P^*, U^*) \vdash \forall \neg p_1(\bar{s}, \theta).
The induction hypothesis states that

\[(P^*, U^*) \models (-G_1, \theta'),\]

where \(\theta'\) is the computed answer substitution corresponding to the SLDN refutation of \(G_1\). Then the whole computed answer substitution is \(\theta = \theta' \circ \theta''\), and obviously,

\[(P^*, U^*) \models \forall \neg p_1(\tilde{s}_1 \theta) \implies (P^*, U^*) \models \forall \neg p_1(\tilde{s}_1 \theta').\]

This, in conjunction with the above induction hypothesis, gives the thesis. Secondly, the derived goal is \(G_1 = (\tilde{q}(\tilde{i}'), G')\theta' \) with \(\theta' = \text{mgu}(\tilde{s}_1, \tilde{i})\), i.e., it is obtained by an ordinary SLD-resolution step using the clause \(\tilde{p}_1(\tilde{i}) :- \tilde{q}(\tilde{i}')\). In this case the induction hypothesis states that

\[(P^*, U^*) \models (-G_1, \theta''),\]

where \(\theta''\) is the computed answer substitution corresponding to the SLDN refutation of \(G_1\). This implies the following two facts:

(i) \((P^*, U^*) \models \forall \neg q(\tilde{i}') \theta' \theta'',\)

(ii) \((P^*, U^*) \models (-G' \theta', \theta'').\)

Since \((P^*, U^*)\) contains the axiom

\[\forall \tilde{y} \tilde{p}_1(\tilde{y}) \iff \exists \tilde{x} (\tilde{y} = \tilde{i}, \tilde{q}(\tilde{i})),\]

from (i) above we obtain

\[(P^*, U^*) \models \forall \neg p_1(\tilde{s}_1 \theta'),\]

which implies

\[(P^*, U^*) \models \forall \neg p_1(\tilde{s}_1 \theta' \theta'').\]

The whole computed answer substitution is \(\theta' \theta''\), and this last statement, in conjunction with (ii) above, gives the thesis.

(6) \(P_{\text{neg}}\) contains the following definition of \(\tilde{p}_1:\)

\[\tilde{p}_1(\tilde{x}) :-\]

and the derived goal is \(G_1 = G'\). The thesis obviously holds from the inductive hypothesis and the fact that \((P^*, U^*)\) contains the axiom \(\forall \tilde{x} (\tilde{p}(\tilde{x}) \iff \text{false}).\)

(7) Finally, if \(\tilde{p}_1(\tilde{s}_1) = \tilde{e}(t_1, t_2)\), let \(G''\) be the goal obtained from \(G\), and \(\theta\) be the computed answer substitution. Then the thesis holds trivially, by the induction hypothesis and the observation that \(t_2 \theta\) does not unify with \(t_2 \theta\), since \(\theta\) is a correct answer substitution for \(\tilde{e}(t_1, t_2)\).

The soundness theorem along with Proposition 5.1 allows us to state the next, more intuitive result

**Corollary 6.1.** Let \(P\) be an FLP, \(P_{\text{neg}}\) its transformation, and \(G\) the goal :- \(\tilde{p}(\tilde{i})\). If \(\theta\) is a computed answer substitution for \(G\), then \((P^*, U^*) \models \forall \neg p(\tilde{i}) \theta\) (or equivalently \(p(\tilde{i}) \theta\) has only finitely failed SLD derivations).
The completeness theorem states that actually SLDN resolution is able to compute answer substitutions for negative theorems of \((P^*, U^*_DCA)\). In what follows we will say "the proof of \(p(i)\) fails" to mean \(p(i)\) has only finitely failed SLD derivations.

**Theorem 6.2.** Let \(P\) be a FLP, \(P_{neg}\) its transformation, \(\bar{g}\) a ground tuple, and \(\bar{s}\) a tuple such that \(\bar{g} \leq \bar{s}\). If \((P^*, U^*_DCA) \models \neg p(\bar{g})\), then \(\bar{p}(\bar{s})\) has an SLDN refutation with computed answer substitution \(\theta\) such that \(\bar{s}\theta \geq \bar{g}\).

**Proof.** Notice that, by Proposition 6.1, \((P^*, U^*_DCA) \models \neg p(\bar{g})\) means that the proof of \(p(\bar{g})\) finitely fails. Hence we can refer to the finite failure of \(p(\bar{g})\) all through the proof.

First of all notice that the theorem trivially holds if \(p(\bar{g}) = eq(g_1, g_2)\), since in this case, by completeness of standard SLD resolution and by definition of \(eq, eq(\bar{s})\) has an SLD derivation (and hence an SLDN derivation) yielding an answer substitution \(\theta\) as in the thesis. The rest of the proof is carried out by induction on the length of the failure of \(p(\bar{g})\).

**Base step:** \(p(\bar{g})\) fails in exactly one step. Then two cases arise:

(i) \(P\) does not contain a clause defining \(p\). Hence \(P_{neg}\) contains the fact \(\bar{p}(\bar{x}) :\leftarrow\), and the SLDN refutation of \(\bar{p}(\bar{s})\) using this fact yields the identity substitution as the computed answer substitution.

(ii) \(P\) contains a clause for \(p\) of the kind \(p(i) :\leftarrow q(i'),\) and moreover, \(p(\bar{g})\) does not unify with \(p(i)\). Then there exists \(i_E \in Not(i)\) such that \(\bar{g}\) is an instance of \(i_E\). Hence \(P_{neg}\) contains the clause

\[
\bar{p}(i_E) :\leftarrow,
\]

\(\bar{p}(\bar{s})\) unifies with \(\bar{p}(i_E)\) via \(\theta\), and, of course, \(\bar{s}\theta \geq \bar{g}\).

**Inductive step:** Assume that the theorem holds for ground atoms whose proof fails in at most \(n - 1\) steps. Let \(p(\bar{g})\) be such that its proof fails in \(n\) steps. Then the following cases arise, depending on the definition of \(p\):

1. \(P\) contains the clause \(p(\bar{x}) :\leftarrow q(\bar{x}), r(\bar{x})\), and either \(q(\bar{g})\) or \(r(\bar{g})\) fails in at most \(n - 1\) steps. Suppose that \(q(\bar{g})\) fails. By definition, \(P_{neg}\) contains the clauses \(\bar{p}(\bar{x}) :\leftarrow \bar{q}(\bar{x}); \bar{r}(\bar{x})\). Then, the following is the first step of an SLDN derivation of \(\bar{p}(\bar{s})\):

\[
G_1 = \bar{q}(\bar{s}) \text{ is derived from } G_0 = \bar{p}(\bar{s}) \text{ via the identity substitution}
\]

Applying the induction hypothesis, we obtain that \(\bar{q}(\bar{s})\) has an SLDN refutation yielding a substitution \(\theta\) such that \(\bar{s}\theta \geq \bar{g}\), which finally yields the thesis.

2. \(P\) contains the clauses \(p(\bar{x}) :\leftarrow q(\bar{x}); r(\bar{x})\), and both \(q(\bar{g})\) and \(r(\bar{g})\) fail in at most \(n - 1\) steps. By definition, \(P_{neg}\) contains the clause \(\bar{p}(\bar{x}) :\leftarrow \bar{q}(\bar{x}); \bar{r}(\bar{x})\). The following is the first step of an SLDN derivation of \(\bar{p}(\bar{s})\):

\[
G_1 = \bar{q}(\bar{s}), \bar{r}(\bar{s}) \text{ is derived from } G_0 = \bar{p}(\bar{s}) \text{ via the identity substitution}
\]

By Proposition 5.1 we can assume that the computation rule selects subgoals from left to right. Hence the evaluation of \(\bar{q}(\bar{s})\) is performed before \(\bar{r}(\bar{s})\) is
evaluated. Applying the induction hypothesis, the SLDN derivation of $\bar{q}(\bar{s})$ yields an answer substitution $\theta'$ such that $\bar{s}\theta' \geq \bar{g}$. Apply again the induction hypothesis, obtaining that $\bar{f}(\bar{s}\theta')$ has an SLDN refutation yielding an answer substitution $\theta''$ such that $\bar{s}\theta'' \geq \bar{g}$, hence the thesis.

(3) $P$ contains the clause $p(\bar{x}) :- q(\bar{x}, \bar{y})$, and the proof of $q(\bar{g}, \bar{y})$ fails in at most $n - 1$ steps. Let $\bar{g}'$ be a ground instantiation for the variables in $\bar{y}$. Then, of course, $q(\bar{g}, \bar{g}')$ fails in at most $n - 1$ steps. Here is the first step of an SLDN derivation for $\bar{p}(\bar{s})$

$$G_i = \langle \bar{s}(\bar{x}, \bar{y}), \text{naf}(q(\bar{s}, \bar{z})) \rangle$$

is derived from $G_0 = \bar{p}(\bar{s})$

via the identity substitution.

Apply the induction hypothesis and obtain that $\bar{q}(\bar{s}, \bar{y})$ has an SLDN refutation with answer substitution $\theta'$ such that $\bar{s}\theta' \geq \bar{g}$. In order to complete the proof, it remains to show that the subgoal $\langle \theta', \text{naf}(q(\bar{s}\theta', \bar{z})) \rangle$ has an SLDN refutation. This is obviously true, since there exists a solution $\theta''$ for herbrand($\bar{s}\theta'$) such that $q(\bar{s}\theta'', \bar{z})$ has only finitely failed derivations. This solution is at most such that $s\theta'' = \bar{g}$, and we know by hypothesis that $q(\bar{g}, \bar{z})$ finitely fails.

(4) $P$ contains the clause $p(\bar{i}) :- q(\bar{i})$. Let $\gamma = \text{mgu}(\bar{g}, \bar{i})$, and $\bar{g}'$ be the ground tuple $\bar{i}'\gamma$. Then $q(\bar{g}')$ fails in at most $n - 1$ steps. But $P_{\text{neg}}$ contains the clause $\bar{p}(\bar{i}) :- q(\bar{i}')$, and there exists $\theta' = \text{mgu}(\bar{s}, \bar{i})$ such that $\bar{s}\theta' \geq \bar{g}$ and $\bar{i}'\theta' \geq \bar{g}'$, since $\bar{s}$ and $\bar{i}$ share the ground instance $\bar{g}$. Then the following is an SLDN-derivation step for $\bar{p}(\bar{s})$:

$$G_i = \bar{q}(\bar{i}'\theta')$$

is derived from $G_0 = \bar{p}(\bar{s})$ via $\theta'$.

By the induction hypothesis, $\bar{q}(\bar{i}'\theta')$ has an SLDN refutation yielding an answer substitution $\theta''$ such that $\bar{i}'\theta'' \geq \bar{g}'$. Of course, $\bar{s}\theta'' \geq \bar{g}$; hence the thesis with $\theta = \theta''$. $\square$

The previous result cannot be extended in order to obtain a completeness result similar to that of SLD resolution for universally quantified positive consequences of program completion. Indeed, a weaker form of completeness holds for SLDN resolution, which is a corollary of Theorem 6.2. This result is stated referring to the notion of term covering, introduced in [18] in order to characterize further operational properties of SLD resolution and negation as failure. Such a notion is, roughly speaking, a generalization of the notion of term instance.

**Definition 6.1.** Let $\bar{i}$ be an $n$-tuple of terms, and $\xi$ a (possibly infinite) collection of $n$-tuples. Then $\xi$ is a covering of $\bar{i}$ iff for each ground instance $\bar{g}$ of $\bar{i}$ there exists $\bar{i}' \in \xi$ such that $\bar{g}$ is an instance of $\bar{i}'$.

**Example 6.1.** Referring to a language providing the constant $a$ and the binary function $f$, the set $\{f(a, a), f(f(x, y), f(z, w))\}$ is a covering of the term $f(x, x)$, and also the set $\{f(a, x), f(y, a)\}$ is a covering of the set $\{f(f(z, w), a), f(a, f(x, y))\}$. 


SLDN refutations actually provide coverings of universally quantified negative theorems of $(P^*, U_{DCA})$. This is stated formally in the following

**Theorem 6.2.** Let $P$ be an FLP, $P_{neg}$ its transformation, and $\bar{i}$ and $\bar{i}'$ tuples of terms such that $\bar{i}' \leq \bar{i}$. Suppose that $(P^*, U_{DCA}) \models \forall \bar{x} \rightarrow p(\bar{i})$; then $\bar{p}(\bar{i})$ has SLDN refutations yielding a set of computed answer substitutions $\{\theta_i\}$ such that $\{i\theta_i\}$ is a covering of $\bar{i}'$.

**Proof.** $(P^*, U_{DCA}) \models \forall \bar{x} \rightarrow p(\bar{i})$ implies $(P^*, U_{DCA}) \models \neg p(\bar{i}')$ for each ground instance $\bar{g}_i$ of $\bar{i}'$. Hence, by Theorem 6.2, $\bar{p}(\bar{i})$ has an SLDN refutation yielding a computed answer substitution $\theta_i$ such that $i\theta_i \geq \bar{g}_i$. The collection $\{i\theta_i\}$ is, of course, a covering of $\bar{i}'$. □

### 7. IMPLEMENTATION ISSUES

Our approach to negation has been carried out with respect to factorized logic programs for the sake of simplifying the formal treatment. In fact this allowed us to focus on the negation of basic combinators for logic programming, as shown in Section 4. Nevertheless it is possible to devise an alternative definition of the transformation itself which seems much more feasible for an implementation of the whole approach. This alternative technique has been actually implemented in a prototype system [20]. In this section we introduce it via a simple example, and then we give its definition.

First of all, we refer to logic programs satisfying the following two requirements:

(i) each clause is left-linear, i.e. all variables occurring in clause heads are distinct each other;

(ii) within clause bodies, each local variable (if any) occurs within a single literal.

Notice that these requirements are far less stringent then the ones corresponding to factorized logic programs and can be achieved by a straightforward precompilation of any logic program. The basic idea is to apply a technique of program transformation based on the well-known *fold/unfold* rules [27], as shown in the following

**Example 7.1.** Let $P$ be the following logic program, where $p$ stands for the *even* relation over natural numbers:

\[
p(0) :-
\]

\[
p(s(s(x))) :- p(x)
\]

The FLP corresponding to $P$ is the following program $P'$:

\[
p(x) :- p'(x)
\]

\[
p(x) :- p''(x)
\]

\[
p'(0) :-
\]

\[
p''(s(s(x))) :- p(x)
\]

where $p'$ and $p''$ are new predicate symbols introduced in order to meet the factorization requirements. Finally the transformation technique yields the follow-
ing definitions for $\bar{p}$:

\[
\bar{p}(x) : - \bar{p}'(x), \bar{p}''(x)
\]

\[
\bar{p}'(s(x)) : -
\]

\[
\bar{p}''(0) : -
\]

\[
\bar{p}''(s(0)) : -
\]

\[
\bar{p}''(s(s(x))) : - \bar{p}(x)
\]

Apply now all the possible unfoldings of the literals $\bar{p}'(x)$ and $\bar{p}''(x)$ in the first rule, obtaining the two rules

\[
\bar{p}(s(0)) : -
\]

\[
\bar{p}(s(s(x))) : - \bar{p}(x)
\]

Notice that this definition of $\bar{p}$ is actually the clausal definition of the relation $odd$ over natural numbers. It is worth noting that the repeated application of the unfold rule does not affect the meaning of negated programs with respect to the original predicate symbols.

It can be shown that the same result can be obtained by applying suitable transformation rules to the unfactorized program [20]. The basic transformation, which yields the negation of a clause, is defined as follows:

**Definition 7.1.** Let $C$ be a clause of the form

\[
p(i) : - p_1(i_1), \ldots, p_k(i_k)
\]

such that $i$ is unrestricted and each local variable (if any) occurs within a single literal of the body. Then $\text{NegC}(C)$ is the following set of rules:

\[
\bar{p}(s) : -
\]

\[
\bar{p}(i) : \forall z_i. \bar{p}(i_i)
\]

for each $s \in \text{Not}_L(i)$

for each $i = 1, \ldots, k$, with $z_i = \text{vars}(i_i) \setminus \text{vars}(i)$.

It should be clear that $\bar{p}(i) : \forall z_i. \bar{p}(i_i)$ has to be interpreted as $\bar{p}(i) : \bar{p}(i_i)$ whenever no local variables $z_i$ occur in $i_i$.

Let us now introduce an operator which combines together formulae obtained in different applications of $\text{NegC}$.

**Definition 7.2.** Let $R_1$ and $R_2$ be two formulas of the kind

\[
A_1 : - B_1
\]

\[
A_2 : - B_2
\]

resulting from the applications of $\text{NegC}$. Then the formula $R_1 @ R_2$ is defined as

\[
A_1 \theta : - (B_1, B_2) \theta
\]

if there exists $\theta = \text{mgu}(A_1, A_2)$.

Notice that the operator $@$ has interesting properties per se. As shown in [21], $@$ is a commutative, associative, and idempotent operator. Finally, the overall transformation can be carried out as follows.
Definition 7.3. Let $P$ be a logic program, and $\text{CL}(p)$ be the set of clauses of $P$ defining the predicate $p$. Then

(i) $\text{Neg}(p) = \{ \tilde{p}(\bar{x}) : - \}$ if $\text{CL}(p) = \emptyset$,

(ii) $\text{Neg}(p) = \{ R \mid R = R_1 \circ R_2 \circ \cdots \circ R_n, \quad R_i \in \text{NegC}(C_i) \}$ if $\text{CL}(p) = \{ C_1, \ldots, C_n \}, \ n > 0$.

Finally, $\text{Neg}(P)$ is the collection of $\text{Neg}(p)$ for each predicate $p$ in $P$.

It is worth noting that the application order of $\circ$ in case (ii) does not affect the result, in view of its properties, and that the resulting formula $R$ is defined iff there exists mgu$(A_1, \ldots, A_n)$ where $A_i$ is the head of $R_i$.

Roughly speaking, the operator $\text{NegC}$ represents the potential contribution that each clause defining $p$ gives to the definition of $\tilde{p}$. On the other hand, $\text{Neg}$ combines the contributions of each clause in order to devise the actual definition of $\tilde{p}$.

Example 7.2. Let us apply the new transformation technique to the program of Example 7.1:

$\text{NegC}(p(0) :- ) = \{ \tilde{p}(s(z)) :- \}$

$\text{NegC}(p(s(s(x))) :- p(x)) = (\tilde{p}(0) :- \tilde{p}(s(0)) :- \tilde{p}(s(s(x))) :- \tilde{p}(x)).$

Observing that the only applications of $\circ$ yielding a rule for $\tilde{p}$ are

$(\tilde{p}(s(z)) :- ) \circ (\tilde{p}(s(0)) :- ) = \tilde{p}(s(0)) :-$

$(\tilde{p}(s(z)) :- ) \circ (\tilde{p}(s(s(x))) :- \tilde{p}(s(s(x)))) = (\tilde{p}(s(s(y))) :- \tilde{p}(y))$,

the result of $\text{Neg}(p)$ is

$\tilde{p}(s(0)) :-$

$\tilde{p}(s(s(y))) :- \tilde{p}(y)$

which is equivalent to the one obtained in Example 7.1.

Given a logic program $P$, let

$\tilde{P}$ be its transformation obtained by factorization and Definition 4.4;

$G$ be an SLDN goal which contains only predicate symbols $\tilde{p}$ corresponding to predicate symbols $p$ originally occurring in $P$ (in other words, no symbol introduced by factorization occurs in $G$).

The following proposition is proven in [20].

Proposition 7.1. $G$ has an SLDN refutation in $\tilde{P}$ with answer substitution $\theta$ iff $G$ has an SLDN refutation in $\text{Neg}(P)$ with answer substitution $\sigma$ such that $G\theta = G\sigma$ up to variable renaming.
The proof of the above proposition is carried out by showing that SLDN-derivation steps in \( \hat{P} \) combined with applications of the unfold rule correspond to SLDN-derivation steps in \( \text{Neg}(P) \) and vice versa.

8. CONCLUDING REMARKS

Some further considerations about SLDN resolution are needed, in particular about the evaluation of naf subgoals of the kind \( \langle \bar{q}(\bar{x}, \bar{y}), \text{naf} q(\bar{x}, \bar{z}) \rangle \). In Example 5.1 we showed that the choice of adopting \( \bar{q}(\bar{x}, \bar{y}) \) as the generator of candidate solutions for \( \bar{x} \) together with the naf literal as the filter of correct solutions does not meet the completeness requirement. In fact, the need of further instantiating candidate solutions has been pointed out along with the introduction of the predicate herbrand, which is able to arbitrarily instantiate (open) terms. One should worry whether there is a real need of using a generator of candidate solutions of arbitrary complexity, as herbrand seems to be. In fact, this point needs further investigations, although we conjecture that, in general, an incorrect candidate solution has to be arbitrarily instantiated in order to find all the correct solutions.

Anyway, it should be pointed out that the use of the herbrand predicate does not allow one to apply the negation-as-failure rule to the transformed program. In fact, the herbrand predicate, when dealing with an infinite Herbrand universe, always diverges. In other words, it is meaningless to talk about the finite failure of SLDN.

Moreover, further research is needed in order to devise a feasible implementation of SLDN, and again, the main concern is about naf subgoals. Of course, the use of the herbrand predicate is computationally infeasible, since it leads to infinite computations in most cases. In fact, given an incorrect candidate solution \( \bar{r} \), suppose that herbrand(\( \bar{r} \)) computes the correct solution \( i \theta \). Then each instance of \( i \theta \) is of course a correct solution too, but obviously none of them is needed in order to achieve the completeness of the evaluation process. Thus, whenever the evaluation of herbrand yields a correct substitution \( \theta \), further evaluations of herbrand which yield instances of \( \theta \) should be cut.

Actually we implemented two versions of SLDN: The first one is the incomplete version which does not adopt any variant of herbrand, although it works in most practical cases [6]. The second one adopts the \text{naf} operator along with the herbrand predicate, exploiting the extralogical features of PROLOG in order to achieve the above behavior, i.e. getting rid of any recursive call to the operator itself whenever a correct solution has been obtained [20].

The transformation technique presented in this paper has been developed within a research project aiming at the definition of a functional metalanguage for logic programming [19, 21]. Roughly speaking, the idea is to equip an ML-like (typed) functional language with a data type designed to represent logic theories. The functional layer is intended to provide some (intensional) operators, such as union, intersection, and negation, able to combine logic theories (similar ideas are sketched in [4]). The \text{intensional negation} operator is essentially defined via the transformation technique proposed in this paper.

We would like to thank Jean-Louis Lassa and Michael Maher for their comments and suggestions on previous versions of this paper; Howard Blair, Giorgio Levi, and Maurizio Martelli for many valuable discussions; and the referees for their careful reviewing.
REFERENCES